Introduction to Attack Patterns

Sean Barnum, Cigital, Inc. [vita[3]]
Amit Sethi, Cigital, Inc. [vita[4]]

2006-11-07 L3 / L, M[5]

This article is the first in a coherent series introducing the concept, generation, and usage of attack patterns as a valuable knowledge tool in the design, development, and deployment of secure software. It is recommended that the reader also review the following articles to fully understand the context and value of attack patterns.

Design patterns are a familiar tool used by the software development community to help solve recurring problems encountered during software development [Gamma 95[6]]. These patterns attempt to address head-on the thorny problems of secure, stable, and effective software architecture and design. Since the introduction of design patterns, many other types of patterns relevant to software have been conceived, including a relatively new construct known as attack patterns [Hoglund 04[7]].

Attack patterns apply the problem-solution paradigm of design patterns in a destructive rather than constructive context. Here, the common problem targeted by the pattern represents the objective of the software attacker, and the pattern's solution represents common methods for performing the attack. Techniques for exploiting software tend to be few and fairly specific [Hoglund 04[8]]. Attack patterns describe the techniques that attackers may use to break software.

The incentive behind using attack patterns is that software developers must think like attackers to anticipate threats and thereby effectively secure their software. Due to the absence of information about software security in many curricula and the traditional shroud of secrecy surrounding exploits, software developers are often ill-informed in the field of software security and especially software exploitation. The concept of attack patterns can be used to teach the software development community how software is exploited in reality and to implement proper ways to avoid the attacks.

Often, security policy also lacks a comprehensive understanding of the issues surrounding software security, as developers have a natural propensity to think in terms of features and functions. Widely accepted and implemented policies that tout encryption as a silver bullet for security problems are an example. Company representatives commonly reassure clients that their data are protected because the database in which they is stored is encrypted. With the hype surrounding firewalls and encryption, it is difficult for the software development community to learn how to actually build secure software. These articles will demonstrate the use of one key tool for effectively building secure software in the absence of any silver bullets.

Terminology

A lot of terminology used in software security has not been standardized. Different publications use different terminology to describe the same concepts and sometimes even use the same terminology to describe different concepts. Furthermore, marketing literature often misuses security-related terms to sell particular products, adding to the confusion surrounding software security. This section attempts to mitigate the issue for the purposes of these articles. It will briefly describe the essential terminology used, which is mostly

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/601-BSI.html (Sethi, Amit)
6. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_gamma95 (Attack Pattern References)
7. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_hoglund04 (Attack Pattern References)
8. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_hoglund04 (Attack Pattern References)

borrowed from *Exploiting Software* [Hoglund 04[9]]. The Attack Patterns Glossary[10] should be consulted for a more complete list of terminology used in these articles.

target software	Target software is software that is the target of an attack.
target host	A target host is the computer or platform that is running the target software of an attack. A host may be attacked through the interfaces provided by the target software or through purely network-based attack mechanisms.
exploit	An exploit is a technique or software code (often in the form of scripts) that takes advantage of a vulnerability or security weakness in a piece of target software.
attack	An attack is the act of carrying out an exploit.
attacker	An attacker is the person or agent that actually executes an attack. Attackers may range from very unskilled individuals leveraging automated attacks developed by others ("script kiddies") to well-funded government agencies or even organized criminals with extensive software backgrounds.
attack pattern	An attack pattern is a general framework for carrying out a particular type of attack, such as a method for exploiting a buffer overflow or an interposition attack that leverages certain kinds of architectural weaknesses. In these articles, an attack pattern describes the approach used by attackers to generate an exploit against software.

Context

Before beginning a discussion on attack patterns, we first need to discuss why attack patterns are important. Attack patterns provide a way for software developers to learn about how their software may be attacked. Armed with knowledge about possible or probable attacks, developers can take steps to mitigate the likelihood or impact of these attacks.

Challenges

Many challenges inhibit the development of secure software. These challenges include

- the actual difficulty of building secure software,
- market forces that favor functionality and time to market over security, and
- a significant knowledge gap between the "black hat"[11] attacking community and the defending software development community with a lack of basic awareness of security issues and solutions.

The Attacker's Advantage

The primary challenge in building secure software is that it is much easier to find vulnerabilities in software than it is to make software secure. As an analogy, consider a bank vault. Its designers need to ensure that it

9. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_hoglund04 (Attack Pattern References)
10. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/590-BSI.html (Attack Pattern Glossary)

is safe against many different types of attacks, not just the seemingly obvious ones. It must generally be safe against mechanical attacks (e.g., using bulldozers), explosives, and safe cracking, to name a few, while still maintaining usability (i.e., allowing authorized personnel to enter, having sufficient ventilation and lighting). This is clearly not a trivial task. However, the attacker may simply need to find one exploitable vulnerability to achieve his or her goal of entering the vault. The attacker may try to access the vault through various potential means, including through the main entrance by cracking the safe combination, through the ceiling, by digging underground, by entering through the ventilation system, by bribing an authorized employee to open the vault, or by creating a small fire in the bank while the vault is open to cause all employees to flee in panic. Given these realities, it is evident that building and maintaining bank vault security is typically much more difficult than breaking into one.

Building secure software has similar issues, but the problem is exacerbated by the virtuality of software. With many systems, the attacker may actually possess the software (obtaining a local copy to attack is often trivial) or could attack it from anywhere in the world through networks. With the ability to attack remotely and without physical access, attacks become much easier. Audit trails may not be sufficient to catch attackers after an attack takes place, because attackers could leverage the anonymity of an unsuspecting user's wireless network or public computers to launch attacks.

Given the greater risks that software faces compared to physical objects, it is essential that software be built with security in mind. To do this, the developers must have a solid understanding of the attacker's perspective to anticipate and thwart expected types of attacks. This is especially true when the assets protected by the software are just as valuable as physical assets protected in bank vaults. Just as bank vaults are built considering all known high-risk attacks that they may face, software should be built considering all applicable known types of attack.

Functionality Over Security

Another challenge is market forces that demand software developers to maximize functionality and minimize time to market. Functionality is what generally sells software, and security is usually treated as an afterthought. Because users do not see most security capabilities, they are not usually considered a priority.

The most successful products tend to be those that offer the most functionality and enter the market before their competitors'. Unfortunately, this holds true for security products such as encryption software, antivirus software, firewall software, etc. The products offering the best functionality are often chosen over the ones that offer the best security. Because of this, more and more systems are being exploited with significant newsworthy consequences. As time passes, the shortsightedness of this approach is becoming clear to the industry, but it will still remain a challenge for many years to come.

The Knowledge Gap

A final central challenge in the area of software security arises from the fact that attackers have been learning how to exploit software for several decades, but the general software development community has not kept up with the knowledge that attackers have gained. This knowledge gap is also evident in the difference of perspective between attackers with their cynical deconstructive view and developers with their happy-go-lucky "you're not supposed to do that" view. The problem continues to grow in part because of the traditional fear that teaching how software is exploited could actually reduce the security of software by helping the existing attackers and even potentially creating new ones. The software development community hoped, in the past, that obscurity would keep the number of attackers relatively small. This assumption has been shown to be a poor one, and some elements of the community are now beginning to look for more effective methods of addressing this problem.

Of course, many other issues pose challenges for software security, but the challenges described here are among the most significant. A basic understanding of the attacker's perspective will help to address these challenges.

Solution

One potential solution to these challenges is using attack patterns to help others understand the attacker's perspective. The black hat community is already well-versed in the techniques used to attack software, but the software development community is not generally educated in the ways in which software is exploited. Attack patterns provide a coherent way of teaching designers and developers how their systems may be attacked and how they can effectively defend them.

A common problem is that software developers try to harden small pieces of software while leaving gaping holes in the big picture. For instance, a developer may use 256-bit AES encryption to secure data but then store the key in the application itself. An attacker will of course choose the easiest way to break software. If an attacker needs the key, he/she will not attempt a brute force attack (computationally infeasible) or cryptanalysis (unlikely to be successful). The attacker will simply obtain the key from the code (very easy).

Likewise, builders of secure physical systems, based on centuries of experience, generally know that attackers always choose the easiest way to achieve their goal. As an analogy, a burglar breaking into a house will not pick the lock(s) on the front door and try to guess the code to the security system if he/she can instead cut the phone line to the house (thus disabling the security system) and break a window to gain access to the inside. Thus, the task of making a house more secure should not involve only better locks and longer security system unlocking codes; they should also involve things like stronger windows and cellular backups for the security system (note that cellular signals also can be jammed, although it is currently not quite as easy as cutting a wire), which can help mitigate known likely attacks. Unless software developers understand similar issues in software security, they cannot effectively build secure software.

Attack patterns help to categorize attacks in a meaningful way, such that problems and solutions can be discussed effectively. Instead of taking an ad hoc approach to software security, attack patterns can identify the types of known attacks to which an application could be exposed so that mitigations can be built into the application.

Another benefit of attack patterns is that they contain sufficient detail about how attacks are carried out to enable developers to help prevent them. Attack patterns, however, do not typically contain inappropriately specific details about the actual exploits to ensure that they do not help educate less skilled members of the black hat community (e.g, script kiddies). Information from attack patterns generally cannot be used directly to create automated exploits.

Of course, attack patterns are not the only useful tool for building secure software. Many other tools, such as misuse/abuse cases, security requirements, threat models, knowledge of common weaknesses and vulnerabilities, coding rules, and attack trees, can help. Attack patterns play a unique role amid this larger architecture of software security knowledge and techniques and will be the focus of these articles.

Background

This section will describe the origin of the concept of attack patterns, provide more detail about the definition of an attack pattern, and discuss some related concepts.

Origins

The concept of attack patterns was derived from the notion of design patterns introduced by Christopher Alexander during the 1960s and 1970s and popularized by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in the book *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma 95[12]]. The book discusses vetted solutions to specific problems encountered in object-oriented software design and how to package these solutions for broad leverage in the form of design patterns. A design pattern captures the context and high-level detail of a general repeatable solution to a commonly occurring problem in software design. It is not a low-level design that can be transformed directly into code; it is a description of how to solve a problem that can be used in many situations. Examples of design patterns

12. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_gamma95 (Attack Pattern References)

include the singleton pattern and the iterator pattern. Discussion of these and other specific design patterns is out of scope for these articles but constitutes recommended reading for anyone desiring a full foundational grounding in the context behind attack patterns.

Since the introduction of design patterns, the pattern construct has been applied to many other areas of software development. One of these areas is software security and representation of the attacker's perspective in the form of attack patterns. The term "attack patterns" was coined in discussions among software security thought-leaders starting around 2001, introduced in the paper *Attack Modeling for Information Security and Survivability* [Moore 01[13]] and was brought to the broader industry in greater detail and with a solid set of specific examples by Greg Hoglund and Gary McGraw in 2004 in their book *Exploiting Software: How to Break Code*.

Since the publication of *Exploiting Software*, several individuals and groups throughout the industry have tried to push the concept forward with varying success. These efforts faced challenges like the lack of a common definition and schema for attack patterns, a lack of diversity in the targeted areas of analysis by the various groups involved, and a lack of any independent body to act as the collector and disseminator of common attack pattern catalogues. These articles, as part of the Build Security In effort sponsored by the U.S. Department of Homeland Security, attempts to provide some coherence of definition and structure. Efforts such as the ongoing DHS-sponsored Common Attack Pattern Enumeration and Classification (CAPEC) initiative will collect and make available to the public core sets of attack pattern instances.

Concept

An attack pattern is an abstraction mechanism for describing how a type of observed attack is executed. Following the pattern paradigm, it also provides a description of the context where it is applicable and then, unlike typical patterns, it gives recommended methods of mitigating the attack. In short, an attack pattern is a blueprint for an exploit. We propose that an attack pattern should typically include the following information:

- **Pattern Name and Classification**: A unique, descriptive identifier for the pattern.
- **Attack Prerequisites**: What conditions must exist or what functionality and what characteristics must the target software have, or what behavior must it exhibit, for this attack to succeed?
- **Description**: A description of the attack including the chain of actions taken.
- **Related Vulnerabilities or Weaknesses**: What specific vulnerabilities or weaknesses (see the glossary[14] for definitions) does this attack leverage? Specific vulnerabilities should reference industry-standard identifiers such as Common Vulnerabilities and Exposures[15] (CVE) number, US-CERT[16] number, etc. Specific weaknesses (underlying issues that may cause vulnerabilities) should reference industry-standard identifiers such as the Common Weakness Enumeration[17] (CWE).
- **Method of Attack**: What is the vector of attack used (e.g., malicious data entry, maliciously crafted file, protocol corruption)?
- **Attack Motivation-Consequences**: What is the attacker trying to achieve by using this attack? This is not the end business/mission goal of the attack within the target context but rather the specific technical result desired that could be leveraged to achieve the end business/mission objective. This information is useful for aligning attack patterns to threat models and for determining which attack patterns from the broader set available are relevant for a given context.
- **Attacker Skill or Knowledge Required**: What level of skill or specific knowledge must the attacker have to execute such an attack? This should be communicated on a rough scale (e.g., low, moderate, high) as well as in contextual detail of what type of skills or knowledge are required.
- **Resources Required**: What resources (e.g., CPU cycles, IP addresses, tools, time) are required to execute the attack?

13. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_moore01 (Attack Pattern References)

- **Solutions and Mitigations**: What actions or approaches are recommended to mitigate this attack, either through resistance or through resiliency?
- **Context Description**: In what technical contexts (e.g., platform, OS, language, architectural paradigm) is this pattern relevant? This information is useful for selecting a set of attack patterns that are appropriate for a given context.
- **References**: What further sources of information are available to describe this attack?

Two examples of attack patterns are provided below [Hoglund 04[18]]:

1. **Pattern name and classification**: Make the Client Invisible
    - **Attack Prerequisites**: The application must have a multi-tiered architecture with a division between client and server.
    - **Description**: This attack pattern exploits client-side trust issues that are apparent in the software architecture. The attacker removes the client from the communication loop by communicating directly with the server. This could be done by bypassing the client or by creating a malicious impersonation of the client.
    - **Related Vulnerabilities or Weaknesses**: CWE–Man-in-the-Middle (MITM), CWE- Origin Validation Error, CWE- Authentication Bypass by Spoofing, CWE- No Authentication for Critical Function, CWE- Reflection Attack in an Authentication Protocol
    - **Method of Attack**: Direct protocol communication with the server.
    - **Attack Motivation-Consequences**: Potentially information leak, data modification, arbitrary code execution, etc. These can all be achieved by bypassing authentication and filtering accomplished with this attack pattern.
    - **Attacker Skill or Knowledge Required**: Finding and initially executing this attack requires a moderate skill level and knowledge of the client-server communications protocol. Once the vulnerability is found, the attack can be easily automated for execution by far less skilled attackers. Skill level for leveraging follow-on attacks can vary widely depending on the nature of the attack.
    - **Resources Required**: None, although protocol analysis tools and client impersonation tools such as netcat can greatly increase the ease and effectiveness of the attack.
    - **Solutions and Mitigations**:
      Increase Resistance to Attack: Utilize strong two-way authentication for all communication between client and server. This option could have significant performance implications.

      Increase Resilience to Attack: Minimize the amount of logic and filtering present on the client; place it on the server instead. Use white lists on server to filter and validate client input.
    - **Context Description**: "Any raw data that exist outside the server software cannot and should not be trusted. Client-side security is an oxymoron. Simply put, all clients will be hacked. Of course the real problem is one of client-side trust. Accepting anything blindly from the client and trusting it through and through is a bad idea, and yet this is often the case in server-side design" [Hoglund 04[19]].
    - **References**: *Exploiting Software: How to Break Code*, p.150 [Hoglund 04[20]].
2. **Pattern name and classification**: Shell Command Injection—Command Delimiters
    - **Attack Prerequisites**: The application must pass user input directly into a shell command.
    - **Description**: Using the semicolon or other off-nominal characters, multiple commands can be strung together. Unsuspecting target programs will execute all the commands. An example may be when authenticating a user using a web form, where the username is passed directly to the shell as in:

      ```
      exec( "cat data_log_" + userInput + ".dat")
      ```

18. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_hoglund04 (Attack Pattern References)

The "+" sign denotes concatenation. The developer expects that the user will only provide a username. However, a malicious user could supply "username.dat; rm –rf / ;" as the input to execute the malicious commands on the machine running the target software. Similar techniques are also used for other attacks such as SQL injection. In the above case, the actual commands passed to the shell will be:

```
cat data_log_username.dat; rm –rf /; .dat
```

The first command may or may not succeed; the second command will delete everything on the file system to which the application has access, and success/failure of the last command is irrelevant.

- **Related Vulnerabilities or Weaknesses**: CWE-OS Command Injection, CVE-1999-0043, CVE-1999-0067, CVE-1999-0097, CVE-1999-0152, CVE-1999-0210, CVE-1999-0260, 1999-0262, CVE-1999-0279, CVE-1999-0365, etc.
- **Method of Attack**: By injecting other shell commands into other data that are passed directly into a shell command.
- **Attack Motivation-Consequences**: Execution of arbitrary code. The attacker wants to use the target software, which has more privilege than the attacker, to execute some commands that he/she does not have privileges to execute.
- **Attacker Skill or Knowledge Required**: Finding and exploiting this vulnerability does not require much skill. A novice with some knowledge of shell commands and delimiters can perform a very destructive attack. A skilled attacker, however, may be required to subvert simple countermeasures such as rudimentary input filtering.
- **Resources Required**: No special or extensive resources are required for this attack.
- **Solutions and Mitigations**: Define valid inputs to all fields and ensure that the user input is always valid. Also perform white-list and/or black-list filtering as a backup to filter out known command delimiters.
- **Context Description**: OS: UNIX.
- **References**: *Exploiting Software* [Hoglund 04[21]].

Note that an attack pattern is not overly generic or theoretical. The following is not an attack pattern: "writing outside array boundaries in an application can allow an attacker to execute arbitrary code on the computer running the target software." The statement does not identify what type of functionality and specific weakness is targeted or how malicious input is provided to the application. Without that information, the statement is not particularly useful and cannot be considered an attack pattern.

An attack pattern is also not an overly specific attack that only applies to a particular application. For instance, "When the PATH environment variable is set to a string of length greater than 128, the application foo executes the code at the memory location pointed to by characters 132, 133, 134, and 135 in the environment variable." This amount of specificity is dangerous to disclose and provides limited benefit to the software development community. It is dangerous because it enables black hats to more easily attack particular software without requiring much thought. It is of limited benefit to the software development community because it does not help them discover and fix vulnerabilities in other applications or even fix other similar vulnerabilities in the same application.

Though not broadly required or typical, it can be valuable to adorn attack patterns where possible and appropriate with other useful reference information such as:

- **Source Exploits**: From which specific exploits (e.g., malware, cracks) was this pattern derived and which shows an example?
- **Related Attack Patterns**: What other attack patterns affect or are affected by this pattern?

- **Relevant Design Patterns**: What specific design patterns are recommended as providing resistance or resilience to this attack, or which design patterns are not recommended as they are particularly susceptible to this attack?
- **Relevant Security Patterns**: What specific security patterns are recommended to provide resistance or resilience to this attack?
- **Related Guidelines or Rules**: What existing security guidelines or secure coding rules are relevant to identifying or mitigating this attack?
- **Relevant Security Requirements**: Have specific security requirements relevant to this attack been identified which offer opportunities for reuse?
- **Probing Techniques**: What techniques are typically used to probe and reconnoiter a potential target to determine vulnerability and/or to prepare for an attack?
- **Indicators-Warnings of Attack**: What activities, events, conditions, or behaviors could serve as indicators that an attack of this type is imminent, in progress, or has occurred?
- **Obfuscation Techniques**: What techniques are typically used to disguise the fact that an attack of this type is imminent, in progress, or has occurred?
- **Injection Vector**: What is the mechanism and format for this input-driven attack? Injection vectors must take into account the grammar of an attack, the syntax accepted by the system, the position of various fields, and the acceptable ranges of data [Hoglund 04[22]].
- **Payload**: What is the code, configuration, or other data to be executed or otherwise activated as part of this injection-based attack?
- **Activation Zone**: What is the area within the target software that is capable of executing or otherwise activating the payload of this injection-based attack? The activation zone is where the intent of the attacker is put into action. The activation zone may be a command interpreter, some active machine code in a buffer, a client browser, a system API call, etc. [Hoglund 04[23]].
- **Payload Activation Impact**: What is the typical impact of the attack payload activation for this injection-based attack on the confidentiality, integrity, or availability of the target software?

Related Concepts

There exist many other concepts and tools related to attack patterns, including fault trees, attack trees, threat trees, and security patterns that are available to the community. It is useful to examine and describe these concepts briefly to reduce confusion between these concepts and attack patterns and so that related literature can be used as a reference when researching or using attack patterns.

Bell Labs developed the concept of fault trees for the Air Force in 1962. It was later applied in a software context in the works of Nancy Leveson [Leveson 83[24]] in the early 1980s. Fault trees provide a formal and methodical way of describing the safety of systems, based on various factors affecting potential system failure. Fault trees are commonly used in safety engineering; the goal of which is to ensure that life-critical systems behave as required when parts of them fail [Vesely 81[25]]. Fault trees have system failure as their root node and potential causes of system failure as other nodes in the tree. Any particular node's "children" represent ways in which the node can "fail." The concept of fault trees is especially helpful for analyzing software for which availability/survivability is a major security concern. Fault trees are a fairly mature concept, and an abundance of literature elaborates on the topic. Fault trees and attack patterns have only a very tenuous relationship. Attack patterns are much more closely aligned with attack trees, a derivative of fault trees, which are described below.

The concept of attack trees was first promulgated by Bruce Schneier, CTO of Counterpane Internet Security. Attack trees are similar to fault trees, except that attack trees are used to analyze the security of systems

24. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_leveson83 (Attack Pattern References)
25. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_vesely81 (Attack Pattern References)

rather than safety. Attack trees provide a formal and methodical way of describing the security of systems based on varying attacks [Schneier 99[26]]. Microsoft uses the term "threat tree" to describe the same concept [Swiderski 04[27]]. An attack tree has the attacker's goal as the root, and the children of each parent node represent conditions of which one or more must be satisfied to achieve the goal of the parent node. In this manner, all paths to the root from the leaf nodes indicate potential attacks.

An attack pattern consists of a minimal set of nodes in an attack tree that achieves the goal at the root node. In a tree with only "or" branches, this consists of all paths from a leaf node to the root node. Such paths are also known as "attack paths." In a tree with some "and" branches, an attack pattern may be a sub-tree of the attack tree that includes the root node and at least one leaf node.

Attack trees and attack patterns are complementary concepts that balance and enhance each other. While attack trees provide a holistic view of the potential attacks facing a particular piece of software, attack patterns provide actionable detail on specific types of common attacks potentially affecting entire classes of software. Details and examples of attack trees can be found in [Schneier 99[28]].

Lastly, another concept related to attack patterns is security patterns. Security patterns consist of general solutions to recurring security problems. A security pattern encapsulates security expertise in the form of vetted solutions to these recurring problems, presenting issues and tradeoffs in the usage of the pattern [Kienzle 01[29]]. Examples include implementing account lockout to prevent brute force attacks, secure client data storage, and password authentication. Because general software developers may not be familiar with security best practices or with security issues, security patterns attempt to provide practical solutions that can be implemented in a straightforward manner. Security patterns also list various tradeoffs in the solutions. Security patterns can be an effective complement to attack patterns in providing viable solutions to specific attack patterns at the design level. As such, it should be noted that security patterns generally describe relatively high-level repeatable implementation tasks such as user authentication and data storage. They are not typically suitable for low-level implementation details such as NULL termination of strings or even very high-level design issues such as client-side trust issues. Hence, they are excellent for describing solutions to programming problems with a security context but they do not demonstrate how to avoid most common software development pitfalls. A security patterns repository is available at SecurityPatterns.org[30]. The repository is not meant to be a comprehensive or most up-to-date list of security patterns.

Further Reading

To learn more about the concept of attack patterns and how they can benefit you, it is recommended that you read the remaining articles in this series. They provide a clear picture of the attack pattern generation process[31] (and thereby a much greater contextual understanding of attack pattern content), as well as how attack patterns can improve security enablement[32] of the software development lifecycle. The series also includes a detailed glossary[33] of terms, a comprehensive references[34] listing, and recommendations for further exploration[35] of the attack pattern concept.

---

26. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_schneier99 (Attack Pattern References)
27. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_swiderski04 (Attack Pattern References)
28. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_schneier99 (Attack Pattern References)
29. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_kienzle01 (Attack Pattern References)
30. http://www.securitypatterns.org
31. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/586-BSI.html (Attack Pattern Generation)
32. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/588-BSI.html (Attack Pattern Usage)
33. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/590-BSI.html (Attack Pattern Glossary)
34. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html (Attack Pattern References)
35. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/589-BSI.html (Further Information on Attack Patterns)

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com